Breaking the Isolation-Freshness Trade-off: Joint Adaptive Storage Optimization for HTAP Systems

Zhenghao Ding
Renmin University, China
dingzh22@ruc.edu.cn

Xinyi Zhang
Renmin University, China
xinyizhang.info@ruc.edu.cn

Chao Zhang
Renmin University, China
cycchao@ruc.edu.cn

Yishen Sun
PingCAP
sunyishen@pingcap.com

Kai Xu
PingCAP
xukai@pingcap.com

Wei Lu
Renmin University, China
lu-wei@ruc.edu.cn

Xiaoyong Du
Renmin University, China
duyong@ruc.edu.cn

ABSTRACT

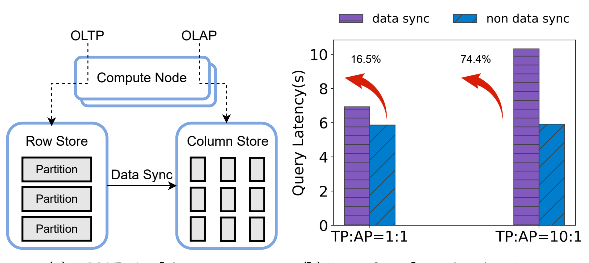
HTAP systems aim to support large-scale transaction processing while preserving real-time analytics over fresh operational data. Achieving this dual goal requires carefully ensuring workload isolation and data freshness. However, this is a challenge that existing systems often struggle to meet, as they rely on static, coarse-grained storage configurations, such as duplicating full data across dual storage model (row and column store) or storing data in a single model. These incur excessive synchronization overhead that degrades data freshness or compromises workload isolation under mixed workloads. To this end, we present Jasper, a joint adaptive storage mechanism that dynamically configures fine-grained storage layouts based on workload characteristics. Jasper performs workload-aware horizontal and vertical partitioning and selectively materializes column store replicas for update-sparse, query-intensive partitions. This design ensures strong workload isolation while minimizing unnecessary data redundancy, significantly reducing synchronization overhead and improving data freshness. We implement Jasper in TiDB and conduct extensive evaluations using both standard benchmarks and a real-world TiDB production workload. Extensive evaluations on both benchmarks and real-world TiDB production workloads show that Jasper cuts workload completion time by 20.43%–40.59%, delivering state-of-the-art performance in balancing isolation and freshness for HTAP systems.

PVLDB Reference Format:
Zhenghao Ding, Xinyi Zhang, Chao Zhang, Yishen Sun, Kai Xu, Wei Lu, Xiaoyong Du. Breaking the Isolation-Freshness Trade-off: Joint Adaptive Storage Optimization for HTAP Systems . PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Godroser/Jasper.

1 INTRODUCTION

HTAP systems support large-scale transaction processing and real-time analytics. The storage model (e.g., row vs. column store) has a significant impact on the performance of different types of workloads [2, 8, 33]. Therefore, existing HTAP systems (TiDB [25], Heatwave [22], SQL Server [32], SingleStore [52]) typically support hybrid storage models and route workloads to the corresponding storage according to their characteristics to achieve high workload isolation (Figure 1a). Specifically, row stores—where data tuples are stored contiguously by row—are used for transactional processing, while column stores—where each column is stored contiguously—are employed to accelerate analytical queries.



(a) HTAP Architecture **(b) Data Synchronization Impact**

Figure 1: HTAP System Characteristics.

Maintaining both row and column stores in HTAP systems introduces data synchronization overhead. To ensure consistency, systems adopt techniques such as delta merge or primary store rebuild [22, 25, 31, 32, 46] to propagate updates from the row store to the column store, which can significantly degrade performance. We conduct a micro-benchmark on TiDB using CH-Benchmark [15] to measure the impact: enabling synchronization increases query latency by 16.5% to 74.4%, with more severe degradation under heavier transactional workloads. This is because TiDB retrieves the latest data on reads [25], and recent updates not yet reflected in the column store introduce additional read latency. These findings highlight the substantial overhead of synchronization over two storage models and prompt a critical question: *is it truly necessary to maintain dual-model storage for all data?*

Storing data in a single model avoids synchronization but compromises workload isolation due to increased contention. This highlights a fundamental trade-off in HTAP systems between isolation and data freshness. Commercial systems like TiDB [25] and ByteHTAP [14] adopt coarse-grained dual-storage designs to reduce contention, but they overlook workload-specific access patterns, leading to redundant data and delayed analytical visibility. In contrast, Peloton [8] employs a single-storage design (i.e., each data tile stored purely in a row or column format) with fine-grained

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

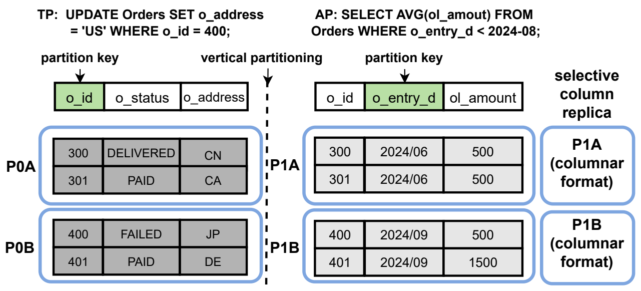


Figure 2: An Example of Adaptive Storage in Table Orders. Write-intensive partition $P0$ is stored by row, while read-intensive partition $P1$ has additional column-store replicas.

tile-based layouts optimized per access pattern, but loses isolation as mixed workloads compete for shared tiles. Proteus [2] introduces a hybrid architecture but lacks systematic partitioning and ignores synchronization overhead, limiting its effectiveness in balancing isolation and freshness.

To better navigate the trade-off between workload isolation and data freshness, we argue that effective HTAP storage layout should move beyond static, coarse-grained storage design toward adaptive and fine-grained optimization. We aim to develop a workload-specific storage mechanism that integrates fine-grained partitioning with selective column store replicas to ensure workload isolation and high data freshness. This mechanism includes both row and column stores. It captures workload characteristics to construct optimal data partitions and selectively configures column store replicas for certain partitions without compromising data freshness. As illustrated in Figure 2, consider an e-commerce scenario with two representative workloads over the *Orders* table: transactions update the address field, and analytical queries calculate the average order amount. Based on workload analysis (e.g., read-write access patterns and query semantics), our mechanism partitions data guided by filter conditions and read-write boundaries while selectively creating column store replicas for analytical hot-spot partitions. For instance, update-dominated partition *P0B* is stored only in row format, avoiding interference with analytics, whereas analytical queries are served from *P1A*, a column-store partition optimized for filtering and aggregation.

However, realizing this fine-grained adaptive storage configuration in practice introduces several key challenges.

C1. Joint optimization of partitioning and selective column replicas. There exists a sequential interdependence between data partitioning and column store configuration. Partitioning defines the scope within which column store replicas can be applied; for example, applying no partitioning (i.e., using a full-table replica) or placing hot and cold data within the same partition can significantly degrade the performance of column store scans. Conversely, column store configuration can influence partitioning decisions: because different storage formats alter query execution costs, a partitioning strategy that is optimal under one storage layout may become suboptimal under another. Moreover, the joint optimization of partitioning and column store presents significant computational overhead. This is due to two key factors: the combinatorial explosion of potential configurations involving multiple columns, and the complex characteristics of HTAP workloads where update/select patterns exhibit cross-column variations and temporal dynamics frequency variations. Prior work [62] proves that data partitioning problem is NP-hard. This complexity further increases when partitioning is combined with column store configuration. These factors result in a massive, entangled design space that makes exhaustive exploration infeasible and efficient heuristics hard to design.

C2. Lightweight yet accurate storage evaluation. Evaluating a candidate storage configuration by materializing it and physically executing queries is prohibitively expensive. Existing "what-if" analysis methods [11, 13, 29, 38, 47] estimate performance under hypothetical configurations using the database optimizer, but suffer from several limitations: First, these methods introduce significant computational overhead due to repeated query plan generation and intrusiveness to optimizer [42, 57]. Second, existing "what-if" analysis rely on query optimizer which suffers from low accuracy in benefit estimation [13, 47]. Besides, they fail to identify and quantify which translational updates are latency-critical for analytical queries, thus ignoring the synchronization overhead in HTAP system. Although deep learning models have gained attention in the field of cost estimation for database systems [34, 62, 63], they introduce significant overhead for model training, as these models require collecting performance data from diverse queries to generalize effectively. Thus, it is both necessary and challenging to develop a lightweight storage evaluation model for HTAP systems that is synchronization-aware and accurate across diverse hardware and workload contexts with minimal training overhead.

C3. Efficient adaptation to dynamic workloads. HTAP workloads are inherently dynamic with frequent shifts in access patterns and concurrency. Static storage optimization strategies could quickly become obsolete, leading to degraded query performance and increased synchronization overhead. While adaptive optimization is necessary, recomputing the entire optimization pipeline, spanning workload profiling, partitioning, and storage model reconfiguration can be prohibitively expensive and highly disruptive to system runtime [17, 24, 48]. Moreover, applying updated configurations could involve non-trivial data movement overhead, especially when transitioning between storage models or repartitioning data. Therefore, an effective storage system must support incremental, low-disruption optimization mechanisms that balance adaptivity with operational efficiency.

To address the aforementioned challenges and balance data freshness and workload isolation for optimal overall performance, we propose Jasper, a joint adaptive storage mechanism that coordinates partitioning and replica materialization in HTAP systems. Built atop a hybrid-store architecture, Jasper partitions the data and selectively materializes column store replicas for primary row partitions to accelerate analytical workloads. First, to efficiently navigate the large design space, we formulate storage optimization as a tree-structured search problem, and design an adaptive search algorithm, MCTS-HTAP. It analyzes access patterns to compute node utilities and action priorities, focusing on more promising configurations, thereby accelerating convergence to the optimal solution (for **C1**). Second, we propose a lightweight storage-aware evaluation model that uses cached plan and adjustment strategy to efficiently generate an execution cost tree for each query under a given virtual storage configuration. Then, rather than directly learning a black-box model to estimate the plan cost, we adopt a more interpretable and modular approach that breaks the problem down into learning individual operator costs and data synchronization overhead and enhances them with storage-aware learned calibration, requiring minimal training overhead (for **C2**). Third, we introduce an incremental reconfiguration mechanism that generates latency-critical reconfiguration action candidates based on detected workload variances, instead of starting from scratch. It models data redistribution costs to balance performance gains and overhead (for **C3**). In summary, we make the following contributions:

- We propose Jasper, a multi-component joint adaptive storage mechanism for HTAP systems. It integrates fine-grained partitioning with selective column store replication to achieve high workload isolation with low data synchronization overhead.
- We design MCTS-HTAP algorithm that transforms the HTAP storage optimization problem into a tree-structured search guided by utility-priority model to avoid combinatorial explosion.
- We propose a lightweight, storage evaluation model for HTAP systems. It generates execution cost tree under new storage configuration by plan caching and adjustment and estimates its cost via individual operator costs and synchronization overhead learning, enhancing accuracy with minimal training overhead.
- We develop an incremental reconfiguration mechanism that dynamically adjusts the storage configuration in response to workload changes. It generates latency-critical reconfiguration actions based on detected workload variances, carefully balancing data redistribution costs with performance gains.
- We implement Jasper in TiDB and conduct extensive experiments. Results show that Jasper reduces end-to-end workload latency from 20.43% to 40.59%, validating its effectiveness.

2 PRELIMINARY

We introduce the basic concepts of data storage in HTAP systems in Section 2.1, propose our problem statement in Section 2.2 and compare representative systems' storage designs to highlight key limitations and motivate our approach in Section 2.3.

2.1 HTAP System

Hybrid Transactional/Analytical Processing (HTAP) systems aim to efficiently support both OLTP and OLAP workloads on a single platform by leveraging the strengths of row and column stores. Row stores handle frequent point queries and updates, while column stores excel at analytical tasks with efficient scans, compression, and vectorized execution.

HTAP systems integrate these models using various architectures, each presenting unique trade-offs in isolation and data freshness. Some systems adopt a dual storage, maintaining row-based storage with column-store replicas (e.g., TiDB [25], ByteHTAP [14], Hyrise-Tired [10]) to improve workload isolation, albeit at the cost of data freshness due to synchronization latency. For instance, TiDB stores all data in TiKV (row store) and maintains columnar replicas in TiFlash, routing queries to either engine based on workload characteristics. Other systems adopt a unified storage. These systems (e.g., Oracle [31], SQL Server [32], Heatwave [22], PolarDB-IMCI [56]) use a primary row store with an in-memory column store, ensuring high throughput but requiring costly delta merges to propagate updates. In contrast, systems like SAP HANA [20, 51], Hyrise [21], Umbra [30, 49] adopt a primary column store with a delta row store, prioritizing analytical performance. While these architectures intrinsically reduce the synchronization overhead between main storage formats, they typically limit OLTP scalability and require extra cost to merge delta data. Each design reflects trade-offs among throughput, freshness, isolation, and scalability [45]. In this work, we target the HTAP systems consisting of a primary row store augmented with column store replicas. we adopt TiDB's hybrid storage model as our foundation due to its balanced architecture and open-source availability, which enables fine-grained storage-level optimization and implementation-level exploration.

To maintain consistency between storage formats, HTAP systems propagate updates from the row store to the column store using asynchronous synchronization, allowing OLTP transactions to commit quickly. While this minimizes write latency, it may delay analytical queries that rely on fresh data, leading to increased query latency and reduced OLAP efficiency. Nevertheless, existing cost estimators [2, 25] overlook synchronization-induced latency, limiting their ability to guide storage layout decisions effectively in hybrid workloads.

2.2 Problem Statement

Definition 2.1. (JORC Problem). Given an HTAP workload window W and a set of tables T, the Joint Optimization for Row and Column stores (JORC) problem seeks an optimal storage layout L that maximizes system performance. The layout $L = \{P, C\}$ is defined as follows:

- **Row Store Configuration**: Partitioning T into disjoint subsets $P = \{P_1, \dots, P_k\}$ via vertical and horizontal partitioning.
- **Column Store Configuration**: Selecting a subset of partitions $C \subseteq P$ to be stored in column store replicas.

Given the inherent challenges in explicitly quantifying workload isolation and data freshness and jointly evaluating their impact, we adopt execution time $T(W)$ as a unified proxy metric that implicitly captures both system characteristics. Execution time naturally reflects the combined impact of these factors: higher freshness reduces staleness-related delays in analytical queries, while stronger isolation mitigates contention and rollback overhead in transactional workloads and reduces execution time. The problem can be formally represented as:

$$\arg\min_{L=\{P,C\}} T(W,L), \quad \text{s.t.} \bigcup_{P_i \in P} P_i = T,\ C \subseteq P. \tag{1}$$

Definition 2.2. (Dynamic JORC Problem). Consider a historical workload window $W = \{q_1, q_2, \dots, q_n\}$, where q represents a query within the workload. The initial storage layout is $L = \{P, C\}$. Let's assume the current workload window has changed to $W' = \{q'_1, q'_2, q'_3, \dots, q'_n\}$, the new storage layout is $L' = \{P', C'\}$. The execution time of workload W' under layout L' is denoted as $T(W', L')$, and the data movement time from L to L' is denoted as $M(L, L')$. Our objective is to identify an new configuration L' that maximizes overall benefit B, where B is defined as the execution time reduction of workload W' between the two configurations minus the data movement overhead $M(L, L')$. The problem can be

Table 1: Comparison of HTAP System Characteristics. Here, ↑ indicates high, ↓ indicates low, and — indicates moderate level.

Method	Storage Model	Column Store	Partitioning	Isolation	Data Freshness
TiDB [25]	Row and Column Store	Table-Level	User-Defined	↑	↓
Peloton [8]	Row or Column Store	Partition-Level	Column Clustering	↓	↑
Proteus [2]	Row and Column Store	Partition-Level	Half Partition	—	—
Jasper	Row and Column Store	Selective	Adaptive Horizontal and Vertical Partition	↑	↑

formally represented as:

$$argmax_{L'} \quad B = T(W', L) - T(W', L') + M(L, L') \tag{2}$$

2.3 Analysis of Related Work

An HTAP storage optimization framework involves two fundamental dimensions: (1) how to determine the optimal combination of partitioning and storage formats, and (2) how to efficiently and accurately evaluate candidate configurations. We first review related work in the two areas, which together motivate our approach.

2.3.1 Storage Layout and Formats. Table 1 summarizes storage strategies in representative HTAP systems. Some systems store each data item in only one format—row or column. Peloton [8] follows this model by assigning a row or column store to each partition based on access patterns. This avoids replication and preserves freshness but offers limited OLTP–OLAP isolation, leading to interference. Other systems [2, 6, 7, 10, 25, 28, 37, 58] maintain both row and column replicas to improve isolation—for example, TiDB keeps dual-format replicas per table—though this adds synchronization overhead and reduces freshness. Proteus [2] allows each partition to use a row store, a column store, or both, but does not model the synchronization cost when dual formats are enabled. Its partitioning strategy is also heuristic, relying on key-range horizontal splits and evenly divided vertical partitions, limiting its ability to balance freshness and isolation.

Beyond storage-centric designs, some work focuses solely on partitioning, independent of storage models. Horizontal partitioning [17, 18, 23, 24, 36, 40, 43, 60] splits tables by rows to improve parallelism and separate hot from cold data, while vertical partitioning [8, 9, 19, 54] divides tables by columns to enhance cache efficiency and enable selective access. Cracking [26] incrementally reorganizes data based on query patterns. However, these layout techniques are generally static or not designed for systems that maintain dual storage formats.

Takeaway for Jasper. These limitations show that neither standalone partitioning nor simple dual-format replication fully unlocks HTAP performance. Replicating both row and column stores improves isolation but introduces heavy synchronization costs and lacks fine-grained control, while static key- or column-based partitioning assumes a row-store design and ignores its interaction with column-store placement. These challenges highlight the need for a unified, workload-aware framework that jointly optimizes partitioning and storage-format decisions in HTAP systems.

2.3.2 Evaluation Models for Storage Configuration. Previous storage-advisor studies fall into two categories: optimizer-dependent and optimizer-independent. Optimizer-dependent methods [4, 12, 13, 26, 29, 41, 47] use "what-if" calls to evaluate storage configurations via the optimizer, but incur high overhead from repeated plan generation, making them impractical for large or real-time workloads. Prior tuning studies [42, 57] have shown that over 75% of the total tuning time is spent querying the optimizer for plan generation under different configurations. Optimizer-independent methods follow two approaches: heuristics [1–3, 9, 35, 48, 50, 57, 60] estimate execution costs based on empirical models, while learning-based methods [24, 62, 63] train predictive models on historical query data. However, heuristics often lack accuracy, and learning-based approaches require large datasets, high computation, and may generalize poorly to new environments.

Takeaway for Jasper. These limitations highlight the need for a storage evaluation approach for HTAP systems that is both efficient and accurate while avoiding frequent optimizer intrusions. In particular, HTAP workloads require evaluating not only query execution cost but also the modeling the interaction effect and synchronization overhead introduced by maintaining row and column replicas—an aspect largely overlooked by prior work [2, 53].

3 OVERVIEW

Figure 3 presents the architecture of Jasper, which comprises three key components: adaptive storage optimization, incremental storage update and storage evaluation model.

Adaptive Storage Optimization aims to select optimal data partition and column replicas based on workload and data characteristics. To jointly optimize data layout, Jasper constructs a structured search space composed of three types of actions: (1) vertical partitioning, which either creates new disjoint column groups or modifies existing ones; (2) horizontal partitioning, which selects a partition key for each vertical partition to split data by rows; (3) column store selection, which determines whether to apply columnar storage to each vertical partition. This space is modeled as a decision tree, where each node represents a partial storage configuration and is generated by applying an action to its parent. To address the combinatorial complexity and interdependence of layout decisions, Jasper leverages a utility-priority model that analyzes workload characteristics to indicate low-impact combinations and capture decision dependencies. Building on this foundation, Jasper employs MCTS-HTAP, a workload-aware search algorithm that extends Monte Carlo Tree Search with utility-guided and priority-driven pruning strategy, enabling the system to focus on promising node and high-impact actions while avoiding unnecessary expansions. As shown in Figure 3, the yellow nodes represent promising nodes, the blue nodes denote unpromising nodes, and the dashed nodes indicate child nodes generated by candidate actions. Instead of uniformly expanding the search tree, MCTS-HTAP selects promising nodes for expansion while simultaneously prioritizing high-value actions and adaptively pruning unpromising ones.

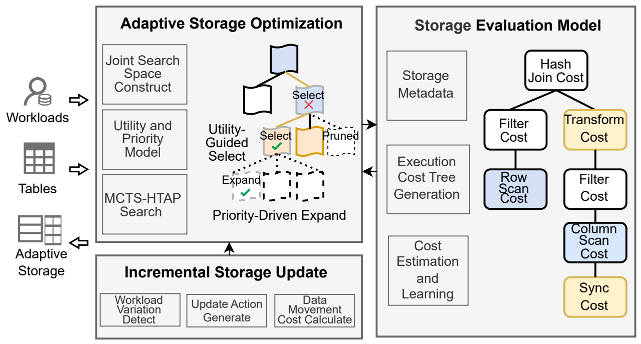


Figure 3: The Jasper Overview.

Storage Evaluation Model efficiently evaluates storage configurations without actual data redistribution. Unlike "what-if" call or existing HTAP optimizers, it is non-intrusive, which avoids spending much time querying the optimizer for plan generation. To better model the execution costs of workloads in HTAP scenarios, we generate execution cost tree to represent the query cost, including operator cost, synchronization cost and data transformation cost (row-column format conversion). The generation of the execution cost tree relies on the query logical plan, instantiating physical operators under the given storage configuration while incorporating costs for data synchronization and data transformation. As shown in Figure 3, the blue *RowScan Cost* and *ColumnScan Cost* are the instantiated operator costs, the yellow *Sync Cost* and *Transform Cost* represent the specific costs of data synchronization and data transformation in HTAP systems. To enhance estimation accuracy to the deployed system and environment, we propose a lightweight learning method to calibrate cost estimation parameters.

Incremental Storage Update adapts to dynamic workloads while eliminating the need for reinitializing the entire optimization pipeline from scratch. Upon detecting a mismatch between current workload and historical access pattern, it first constructs a compact candidate action set, e.g., modifying an ill-suited partition key or disabling unnecessary column-store replicas based on workload variations analysis. To avoid excessive overhead from data redistribute, Jasper models reconfiguration cost of each candidate action to estimate the net benefit. The current state in the policy tree is then incrementally expanded by applying these candidate actions using the MCTS-HTAP policy with maximizing net benefit as objective. By serving as a focused search frontier, the identified compact action set reduces further accelerates the search convergence.

Workflow. Given a workload, Jasper first invokes adaptive storage optimization to explore candidate layouts, guided by fast cost feedback from the storage evaluation model. After establishing a baseline layout, incremental storage update monitors workload shifts and, with support from the evaluation model, applies small but high-impact storage adjustments, ensuring continuous optimization with minimal overhead.

Implementation Jasper is built on TiDB without altering its internal concurrency-control logic and relies on TiDB's MVCC and Raft mechanisms to ensure transactional consistency across row and column replicas. All layout transformations—including repartitioning and column-store adjustments—are performed through an online, non-blocking reconfiguration process that asynchronously builds the new layout, synchronizes updates in the background, and atomically switches to the new version once consistency is achieved. More details can be found in the technical report [27]. Jasper is designed to operate in the background while keeping its behavior observable and explainable through user-facing interfaces. Users can enable or disable Jasper, configure its parameters, such as refitting its evaluation model on recent observations, through built-in UDFs. The storage layout configured by Jasper is persisted as physical metadata within TiDB and is used directly by the query optimizer. Users can inspect its impact on query execution through `EXPLAIN ANALYZE`, including whether partition pruning is applied and which storage formats are used. To support troubleshooting and diagnosis, Jasper logs each layout reconfiguration event, including the trigger time, affected partitions, and old and new layouts.

4 ADAPTIVE STORAGE OPTIMIZATION

We propose MCTS-HTAP, a utility-guided and priority-driven search algorithm designed for HTAP storage optimization. The core idea is to leverage the mixed and highly skewed read-write access patterns of HTAP workloads and their interactions between row and column stores to differentiate states and actions likely to deliver higher performance gains. Based on these insights, a variant search algorithm adaptively prunes low-utility regions, executes high-priority actions early, therefore concentrating depth-wise exploration on promising regions of the search space.

4.1 Adapting MCTS for JORC

We adopt an MCTS-based algorithm because it balances exploration and exploitation, allowing efficient navigation of a large search space without exhaustive enumeration. This section outlines how we adapt MCTS to the JORC problem, with algorithm presented in Sections 4.2 and 4.3.

State represents a candidate configuration of partitioning and column store, with each state corresponding to a node in the tree.

Example 4.1. Given n tables and m columns, each vertically partitioned into p segments, the number of candidate configurations grows factorially as $O\big([2(m/p)!]^{np}\big)$, illustrating a severe combinatorial explosion. Even moderate values of n, m, p make exhaustive search infeasible.

Action represents a decision at a given state—such as applying table partitioning or enabling a column store for a partition. Executing an action moves the system to a new state, creating a child node in the search tree. Each state has a set of executable actions, including all valid partitioning and column-store operations. Partitioning actions are classified as horizontal or vertical; horizontal partitioning selects or adds a column as a partition key within an existing vertical group to split data horizontally.

Example 4.2. A horizontal partitioning action selects a key column (e.g., o_entry_d) for dividing a vertical partition (e.g., o_id, o_entry_d, o_amount). Vertical partitioning moves columns across partitions—e.g., moving $o\_address$ from $P1$ to $P2$. Selective column store enables column store for specific partitions (e.g., $P1$), and this setting is inherited by sub-partitions created later in the search.

Reward is the cumulative gain or benefit obtained by executing a sequence of actions starting from a given state. The reward is estimated by the evaluation model in the simulation phase and it would be updated during the backpropagation phase.

The MCTS search process involves four main steps: (1) **Selection**, where the algorithm traverses the tree from the root to an expandable node using a policy ; (2) **Expansion**, where the node randomly executes an unexplored action to add a new child node; (3) **Simulation**, where a roll-out estimates the reward from the new node; and (4) **Backpropagation**, where the obtained reward is propagated back up the tree.

4.2 Priority and Utility Model

Given the combinatorial explosion and high dimensionality of the search space, traditional MCTS converges slowly and wastes effort exploring low-value configurations—e.g., using rarely accessed columns as partition keys, which offers little pruning or join benefit. To address this, we introduce a priority–utility model that unifies the search space and assigns utility to states and priority to actions based on workload analysis.

Definition 4.3. (Priority). Priority represents the execution preference of an action, denoted by $\mathcal{P}(S, a_i)$. It reflects the expected benefit of executing action a_i in the given state S. Action with a higher priority is more likely to be executed. During node expansion, there are often multiple candidate actions, each leading to different rewards. We use priority to preferentially explore actions that are more likely to yield higher rewards.

Definition 4.4. (Utility). Utility measures the potential exploration benefit of a state (i.e. node). We define the utility of a node as the average priority of its remained executable actions. The utility of state S is given by:

$$U(S) = \frac{1}{n} \sum_{a_i \in A_e} \mathcal{P}(S, a_i) \tag{3}$$

Here, A_e denotes the set of remained executable actions, and n represents the number of such actions. As node S is progressively expanded, its utility $U(S)$ tends to decrease, reflecting the diminishing number of remaining high-priority actions. It quantifies the expansion potential of a node: a higher utility indicates the presence of high-gain actions that have not yet been executed, suggesting that the node is more promising for expansion.

Priority and utility are determined by workload characteristics, particularly column-level read/write frequencies. Write-intensive columns should remain in row store, while read-intensive ones benefit from column store. Thus, actions involving read-heavy columns for partitioning or column store receive higher priority.

Workload-Aware Prioritization. We compute the priority of each action based on the access characteristics of its associated columns. Let $\mathcal{W}$ denote the workload and C the set of columns. For a column $c_i \in C$ and transaction $T_k \in \mathcal{W}$, let $c_i^{k,r}$ and $c_i^{k,w}$ denote the read and write frequencies of c_i in T_k, respectively. The read/write frequency is computed at the operator level. A column's read frequency increases when the associated operator performs a full-table scan, range scan, aggregation, or join. A column's write frequency increases when the operator corresponds to an update, insert, or delete action. We define the priority of column c_i as follow:

$$p(c_i) = \sum_{T_k \in W} (c_i^{kr} - c_i^{kw}), \tag{4}$$

where higher priorities are assigned to read-intensive columns, while write-intensive columns receive negative values. To facilitate subsequent computations, we normalize these priority values to the range $[0, 1]$, denoted as $\hat{p}$.

Building on the fine-grained column-level modeling, we compute the priority of each action based on the priorities of the associated columns. Let $C.a$ denote the set of columns involved in action a. For example, in the case of a partitioning action, $C.a$ refers to the columns used as partition keys; for a column store action, it denotes the columns contained within the corresponding partition. We define the priority of the i-th action a_i in state S as follows:

$$\mathcal{P}(S, a_i) = \frac{\sum\limits_{c_j \in \{C.a_i\}} \hat{p}(c_j)}{\sum\limits_{a_k} \sum\limits_{c_j \in \{C.a_k\}} \hat{p}(c_j)} \tag{5}$$

Here, a_k denotes the set of all executable actions in state S. Notably, an action's priority is independent of the specific state in which it occurs, as it is derived solely from workload analysis. Each state may have multiple executable actions—some of which have already been expanded into child nodes, while others remain unexplored.

4.3 MCTS-HTAP Search

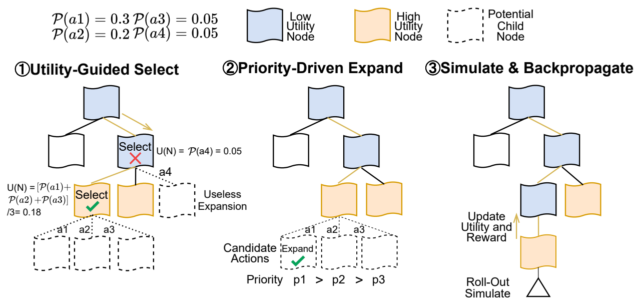
We summarize the core steps of MCTS-HTAP—utility-guided selection, priority-driven expansion, and simulation with backpropagation—as shown in Algorithm 1. This design directs the search toward promising configurations while pruning low-value regions, reducing the search space and accelerating convergence.

Step 1: Utility-Guided Selection. This step selects the most promising node using a utility-guided strategy (line 9). Starting from the root (a configuration without partitioning or column store), the algorithm traverses the tree by prioritizing high-utility nodes, enabling deeper exploration of promising paths and early pruning of low-value ones. A node is expanded only if its utility exceeds a threshold θ; otherwise, its children are examined. The threshold gradually decreases to ensure full exploration. When choosing among child nodes, the algorithm uses UCB1 to balance exploration and exploitation, selecting the node with the highest UCB1 score. UCB1 is a bandit-based strategy that selects the child node with the maximal estimated average reward and an uncertainty term that decreases with the number of visits (defined in Line 12). $N(node)$ is the total number of visits to $node$, and $N(node, a_i)$ is the number of visits to action a_i in $node$. c is the exploration constant (typically set to 2 or determined through parameter tuning). This process repeats until a high-utility node or a leaf is reached.

Example 4.5. As shown in Figure 4, traditional MCTS would still expand a node whose action $a4$ has a low priority (0.05), wasting exploration. MCTS-HTAP instead checks the node's utility (0.05), skips it because it is below the 0.1 threshold, and moves to the child with the highest UCB1 score. That child, whose actions $a1$, $a2$, and $a3$ yield a utility of 0.18, exceeds the threshold and is selected.

Algorithm 1: MCTS-HTAP

```
1  Function MCTS-HTAP($root$, $N$)
       Input : $root$: initial node; $N$: number of iterations
       Output : Best node
2      for $i \leftarrow 1$ to $N$ do
3          $node \leftarrow root$;
4          $node \leftarrow$ Utility-Guided Select($node$);
5          $node' \leftarrow$ Priority-Driven Expand($node$);
6          $reward \leftarrow$ Simulate($node'$);
7          Backpropagate($node'$, $reward$);
8      return node with highest average reward;
9  Function Utility-Guided Select($node$)
10     while $node$ is a non-leaf node do
11         if $U(node) < \theta$ then
12             $node \leftarrow \arg\max_i \left(\bar{R}(node, a_i) + c \cdot \sqrt{\frac{\ln N(node)}{N(node,a_i)}}\right)$;
               // Skip expanding the node
13         else
14             return node; // Expand the node
15     return node;
16 Function Priority-Driven Expand($node$)
17     $Actions = Get\_Executable\_Actions(node)$;
18     for $action$ in $Actions$ do
19         $Priority\_Assign(action)$;
20     $action \leftarrow Priority\_Choose(Actions)$;
21     $node' \leftarrow node.Take\_Action(action)$;
22     return node';
```



$\mathcal{P}(a1) = 0.3$ $\mathcal{P}(a3) = 0.05$
$\mathcal{P}(a2) = 0.2$ $\mathcal{P}(a4) = 0.05$
Low Utility Node
High Utility Node
Potential Child Node
①Utility-Guided Select
②Priority-Driven Expand
③Simulate & Backpropagate
Select
U(N) = $\mathcal{P}$(a4) = 0.05
a4
U(N) = [$\mathcal{P}(a1)+\mathcal{P}(a2)+\mathcal{P}(a3)$]/3= 0.18
Select
Useless Expansion
a1
a2
a3
Candidate Actions
Expand
a1
a2
a3
Priority p1 > p2 > p3
Update Utility and Reward
Roll-Out Simulate

Figure 4: MCTS-HTAP Search.

Step 2: Priority-Driven Expansion. This step generates a new node by applying an executable action given the selected node (line 16). However, a node may possess numerous executable actions, each with different exploratory value. Exhaustively enumerating all possible actions is both computationally expensive and often unnecessary. To address this, Jasper adopts a priority-driven approach rather than selecting actions uniformly at random. For each node, we first identify its set of executable actions and compute the priority of each action as described in Section 4.2. To select an action for expansion, the algorithm invokes Priority_Choose, where the selection probability of each action is proportional to its priority. This mechanism favors actions involving read-intensive columns for partitioning or columnar storage, while write-intensive columns are more likely to be retained in row store.

Example 4.6. As shown in Figure 4, traditional MCTS randomly expands one of three candidate actions, even if its priority is only 0.05. In contrast, MCTS-HTAP computes action priorities and selects an action probabilistically according to its priority.

Step 3: Simulation and Backpropagation. Estimating a new node's long-term benefit is challenging, as it depends on future decisions. MCTS-HTAP addresses this via simulation: once a child node is expanded, a random sequence of actions is taken until a terminal state is reached. The terminal state refers to reaching a predefined maximum depth. The evaluation model (Section 5) computes the reward as the estimated reduction in workload execution cost, This reward is propagated backward along the path traversed during selection and expansion, and updates visit counts, cumulative rewards, and utility. After a set number of iterations, the node with the highest average reward is selected as the final configuration.

4.4 Incremental Storage Adjustment

When workload evolves, the existing storage layout may become suboptimal. However, re-running the full optimization from scratch introduces significant overhead and incurs high cost of data movement. To address this, we propose an incremental update mechanism that restricts the search space based on detected workload variations and incorporates data movement cost into the decision process. Our method comprises two steps: Variations-Driven Candidate Generation and Movement-Aware MCTS-HTAP Search.

Variations-Driven Candidate Generation. It uses workload windows to detect workload variations, defined as changes in the access frequency of columns. A historical window (W) records the workload that led to the current configuration, while a sliding window (W') captures the current workload. If the relative deviation between W' and W exceeds a deviation threshold, Jasper generates candidate update actions on columns with large access-pattern shifts, such as modifying their partition keys or storage formats. It then performs a fast MCTS-HTAP search over this restricted candidate space, avoiding the overhead of exploring the full action space. A new configuration is applied only if the expected performance gain outweighs the data-movement cost. Upon reconfiguration, W is updated with the latest workload. Drastic workload or schema changes are out of scope.

Movement-Aware MCTS-HTAP Search. To ensure practical and efficient updates, we jointly consider both the expected performance improvement and the movement overhead when selecting a new configuration. We estimate the data movement cost using a hybrid cost model. For the affected data, we approximate its movement cost as the sum of (1) the scan cost of the affected data and (2) the I/O cost of loading it into the new layout. Both components are measured in time for cost alignment. The scan cost is estimated using the evaluation model in Section 5. To estimate the I/O overhead, we construct a predictive model that captures the empirical relationship between data volume, I/O bandwidth, and observed load time. Specifically, we apply linear regression over historical traces to fit a cost function of the form:

$$T_{IO} = \alpha \cdot \frac{V_{data}}{Bandwidth} + \beta \tag{6}$$

where T_{IO} represents the I/O cost, V_{data} and $Bandwidth$ refer to the data volume and I/O bandwidth respectively, α and β are the learned parameters.

We employ MCTS-HTAP to search for the updated storage configuration L' that maximizes B. The search initializes from the current configuration L with the search space restricted by the selected candidate actions. Referring to Section 4.2, we calculate the priority of the current action and the utility of the tree nodes. Considering only adjusted data requires data movement, we denote the corresponding partitions and column stores as P and C respectively, and their associated data volume as $V_{\Delta P}$ and $V_{\Delta C}$. This allows the benefit of adjusting the layout, B, to be rewritten as:

$$B(L, L') = T(W', L) - T(W', L') + \alpha \cdot \frac{V_{\Delta P} + V_{\Delta C}}{Bandwidth} + \beta + T_{scan} \quad (7)$$

where T_{scan} is the scan time of affected data. In incremental adjustment, MCTS-HTAP uses $B(L, L')$ as the reward for each node. The selection, expansion, simulation and backpropagation step follow the description in Section 4.3. After a fixed number of iterations, the configuration L' with the highest average B value is selected as the new configuration.

5 STORAGE EVALUATION MODEL

We introduce the execution cost tree (Section 5.1) to enable optimizer-independent and multi-dimensional cost estimation for HTAP workloads. We also employ a lightweight learning mechanism to enhance estimation accuracy (Section 5.2). To illustrate this approach, we use the query optimization mechanism in TiDB as an example, but the concept is generalizable to other database systems.

Definition 5.1. (Execution Cost Tree). The execution cost tree is a hierarchical representation of a query plan that models the execution cost of each operator independently. Compared to a traditional physical plan tree, it additionally incorporates data synchronization and data transformation costs, making it well-suited for the complex cost dynamics of HTAP systems.

5.1 Execution Cost Tree Generation

This section discusses how to efficiently generate execution cost tree under a given storage configuration by maintaining the storage metadata, caching logical plan, and deriving the cardinality.

Storage Metadata Maintenance. To support fine-grained cost analysis in hybrid storage environments, we maintain a unified metadata schema that captures both partitioning and physical storage characteristics, forming the foundation for accurate cost estimation. *Partition Metadata* includes the partitioning information for each table, such as horizontal partitioning keys and vertical partitioning structures. For each partition, it records statistics such as the key range (min, max) and shape of each partition (the number of associated rows and columns). *Row/Column Store Metadata* includes the type of storage model (row or column) of each partition. Given a storage configuration, Jasper updates the metadata accordingly, deriving partition statistics from column histograms without requiring actual data redistribution.

Logical Plan Tree Caching. To eliminate redundant optimizer invoke, we cache logical query plans (preserving join orders, filters, etc.) and dynamically instantiate physical operators (e.g., *TableScan* or *HashJoin* variants) based on storage configurations. For instance, the *Scan* operator has two variants depending on the storage format: a scan over the row store and a scan over the column store. For *Join* operator, we follow the optimizer's default criteria, using *HashJoin* for most cases and switching to *SortMergeJoin* when sorting is involved. Specifically, the storage layout affects *HashJoin* placement. If the two input tables use different storage models, the row store is typically assigned as the build side due to its efficiency in hash table construction, while the column store is assigned as the probe side to leverage its high columnar scan capabilities. If both tables share the same storage model, the larger table is assigned to the probe side to minimize memory consumption and improve performance.

Partition-driven Cardinality Derivation. Partitioning configurations directly influence query access patterns. While predicate selectivity remains logically unchanged, the actual scan scope, i.e., the number of rows and columns accessed, can vary depending on how data is physically partitioned. This, in turn, alters the cardinality of *Scan* and *Selection* operators. Jasper prunes partitions to narrow the physical scan scope, but the Scan output cardinality remains logically determined by the query predicates. Therefore, given a storage configuration, Jasper re-evaluates predicate coverage based on the updated partition ranges and adjusts the cardinality for cost estimation to reflects the accessed data change.

Execution Cost Tree Generation. Based on the storage metadata, cached logical plan and derived cardinality, Jasper generates an execution cost tree for each query under a given storage configuration. The execution cost tree incorporates three distinct cost components: (1) physical operator execution cost, (2) data synchronization cost, and (3) data transformation cost. Physical operators are instantiated directly from the cached logical plan while preserving the original query structure. The operator cardinality is derived based on partition characteristics. For data synchronization, we account for potential latency when *ColumnScan* operators access recently modified data, as columnar stores may require synchronization with the latest updates. This synchronization overhead is explicitly modeled in our cost estimation. Data transformation cost arises when perform join with columnar data, which necessitates format conversion from columnar to row format. We explicitly capture this through our *TransformCost* in the cost tree.

Example 5.2. Figure 5 illustrates the construction of the execution cost tree. Table A uses row format and is horizontally partitioned into four parts by key $a1$, while Table B is stored in both row and columnar formats—vertically split into two segments, with one segment further partitioned by key $b1$. The query's logical plan tree, which scans and joins Tables A and B, is cached. Based on storage metadata, scan operators are instantiated as *RowScan Cost* and *ColumnScan Cost*, with *Sync Cost* added as needed. Partition pruning identifies partitions $P1$ and $P2$ of Table A and $P3$ and $P4$ of Table B as the scan scope, with cardinalities derived from metadata. The join is modeled as a HashJoin, with Table A as the build side and Table B as the probe side, incorporating a *Transform Cost* for pre-join data conversion.

5.2 Cost Estimation and Learning

Jasper estimates the cost for each node (i.e. operator) in the execution cost tree and aggregates them as the total execution cost. To

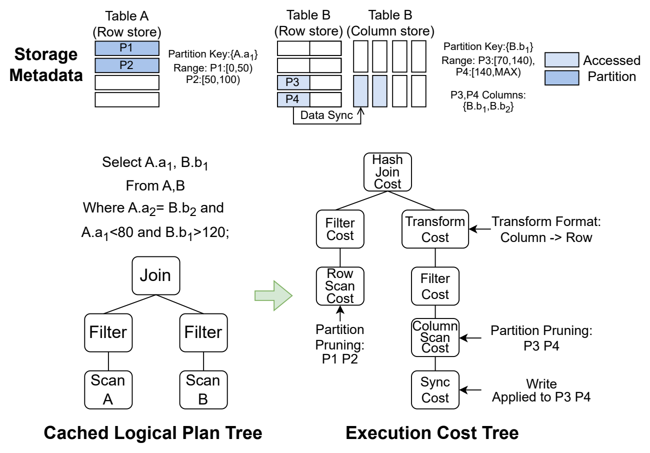


Figure 5: Execution Cost Tree Generation.

improve the accuracy, we calibrate related parameters specific to the storage configurations.

Operator Cost. Building on metadata and cardinality derivation, Jasper predicts the cost of individual node utilizing formula-based model with calibrated parameters. It is shown that formula-based models incorporate prior expert knowledge, making them robust and comparable to neural-network-based models when properly calibrated for storage engines and hardware [59]. Specifically, we derive engine-specific cost functions based on the built-in formulas in TiDB for estimating the common physical operators. For example, For the *Scan* operator, its cost function is defined as:

$$Cost_{scan} = rows \cdot log(rowsize) \cdot scanfactor \tag{8}$$

where *rows* represents operator cardinality, *rowsize* is the width of all columns in the relevant partitions for row stores, while for column store it includes only the queried columns. *Scanfactor* is an engine-specific parameter that typically has a larger value for row stores and a smaller value for column stores. To account for the specific storage engine's influence, we set different parameters for each storage model. More cost functions are available in our technical reports [27].

Data Synchronization Cost. We define synchronization cost as the query delay caused by waiting for the updated data to be synchronized in HTAP systems. While updates are applied to the column store asynchronously, the replication lag introduces extra read latency for analytical queries. It introduces non-trivial read latency as the intensity and frequency of transactional updates increase. However, quantifying this synchronization overhead is inherently challenging due to the heterogeneity of synchronization mechanisms across HTAP systems. Different implementations exhibit varying cost characteristics, making it difficult to generalize a deterministic cost model for accessing intermediate state. To address this issue, we use the estimated volume of synchronized data accessed by analytical queries as a proxy for synchronization cost and learn the empirical relationship between synchronization volume and increased query latency.

The volume estimation consists of the following steps. First, we identify the data that require synchronization by analyzing the transactional workload to identify the set of tables and columns that have been recently modified. Next, the volume of data synchronization is estimated based on each transaction's data access size and its concurrency using catalog metadata such as column-level storage size. Finally, for each analytical query, we examine whether any of its accessed columns intersect with the previously identified TP-updated columns. If such an intersection exists, we consider the query affected by synchronization and attribute additional synchronization cost accordingly. Above all, the data synchronization cost is summarized as:

$$Cost_{sync} = \begin{cases} \alpha \cdot \sum\limits_{t_i \in T} (rowsize_i \cdot con_i) & \text{data intersected} \\ 0 & \text{no data intersected} \end{cases} \tag{9}$$

where α is the parameter, T is the set of intersected OLTP transactions, $rowsize_{t_i}$ is the i-th intersected transaction's data access size, con_{t_i} is the transaction concurrency.

Transform Cost. Data transformation refers to converting columnar data into row format, which involves scanning the relevant columns and the corresponding rows. Thus, it is evident that the cost of data transformation depends on both the column size and the cardinality. Therefore, similar to the *Scan* operator, we define the cost formula for data transformation as follows:

$$Cost_{trans} = rows * log(columnsize) * transfactor \tag{10}$$

where *rows* represents the cardinality, *columnsize* is the sum of related column sizes, *transfactor* is the transform parameter. We derive the specific parameter values based on historical data.

Learning Parameter Calibration. To account for the specific storage engine's influence, we set different parameters for each storage model. Besides, rather than relying on a generic cost model for each engine, we empirically fit these parameters to the target deployed system and environment. These parameters are calibrated using operator-level execution latency data to ensure more accurate cost predictions. Based on the collected data, a least-squares fitting method is applied to infer the optimal parameter values that minimize the discrepancy between estimated and observed latency. This calibration process is lightweight and offline, requiring only a small profiling samples, ensuring minimal overhead while maintaining robustness and portability across different environments.

6 EVALUATION

In this section, we present our experimental evaluation that demonstrates the effectiveness of Jasper storage adaption and how it significantly improves the HTAP system performance. We compare with other HTAP architectures, and discuss the generality of our method in the technical report [27] due to limited paper space.

6.1 Setup

We deploy and conduct experiments using the open-source version of TiDB v8.1.0 [25]. TiDB designs two separate storage engines, a row store (TiKV) and a column store (TiFlash), and uses raft as replication algorithm to synchronize the data between the two storage engines. It asynchronously replicates raft logs to transform row format to column format. We deploy TiDB with 3 TiKV nodes and 1 TiFlash node distributed across two physical machines. Each machine contains two Intel(R) Xeon(R) Gold 5220 CPUs and 128 GB memory, where a 1 Gbps network connects two nodes. The operating system is CentOS 7.9. We construct multiple clients that connect to TiDB via *mysql.connector*.

6.1.1 Baselines. We compare Jasper with state-of-the-art HTAP storage advisors, namely Proteus [2], Peloton [8], as well as TiDB's default setting. To demonstrate the advantages of joint optimization, we also compare with a two-stage optimization approach. For a fair comparison, we carefully implement these methods on TiDB.

Proteus. An HTAP database system that autonomously changes its storage layout to optimize for mixed workloads. It first vertically splits the tables and partition the data based on the primary key, then selects the storage models for each partition based on cost functions without considering data synchronization overhead.

Peloton. An HTAP database system that assigns each data partition either column store or row store based on their query accessing pattern. It first applies horizontal partitioning based on primary keys, followed by vertical partitioning via column clustering, where columns with strong correlations are grouped together. Read-heavy partitions are stored in columnar model while the remaining partition is stored in a row model.

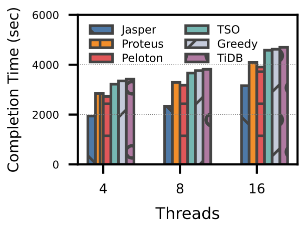
Two-Stage Optimization (TSO). We implement two independently optimized techniques from commercial systems, i.e., Redshift's partitioning techniques [43] and Oracle's column store configuration [31] in TiDB without joint coordination. Redshift models tables and join relationships as a graph, and uses greedy expansion based on the weights of nodes and edges to determine partition keys. Oracle populates frequently read data in columnar format.

Greedy Optimization (Greedy). At each step, it uses the evaluation model to assess the immediate performance gain of all possible actions and greedily executes the one with the highest reward.

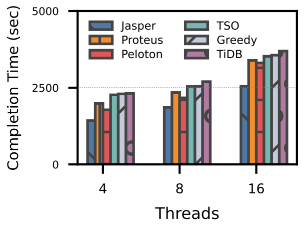
TiDB. TiDB's default setting that partitions the data based on the primary key without vertical partitioning. It maintains both row store and column store replicas for all the partitions.

6.1.2 Workload. We conduct experiments using the CH-Benchmark [15], Twitter [16], HyBench [61] and a real-world workload from TiDB OSS for HTAP systems. The CH-Benchmark is a hybrid transactional and analytical processing (HTAP) benchmark that combines elements of TPC-C and TPC-H to evaluate a system's ability to handle mixed workloads. We generate a dataset of 50GB in size, which includes 200 warehouses. The Twitter workload simulates a social networking application that contains relationships between users, tweets, and their followers. The workload includes six OLAP transactions and three OLTP transactions following previous work [2]. We generate a 20GB dataset. The HyBench simulates a real-world finance application and test QPS, TPS, and their hybrid performance. The OSS workload is generated by an open-source project OSS Insight [44] and encompasses a wide range of hybrid tasks, including analysis, updates, trend detection, and rankings of open-source software. It contains 2 TB of data, 6 tables and more than 8 billion rows of GitHub event records.

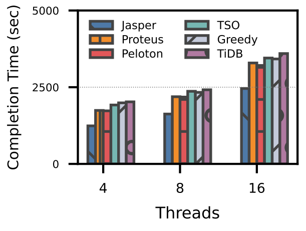
To simulate various hybrid workload scenarios, we construct three types of workload: *OLTP-Heavy*, *Balanced*, and *OLAP-Heavy*. In the OLTP-Heavy workload, the ratio of transactional (TP) operations to analytical (AP) queries is 10:1. The Balanced workload maintains an equal ratio of 1:1 between TP transactions and AP queries, where the OLAP-Heavy workload emphasizes analytical processing with a TP to AP ratio of 1:10.



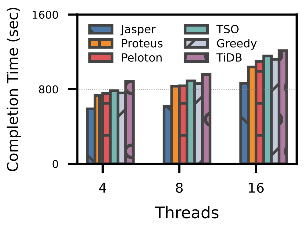
(a) CH-benchmark OLTP Heavy



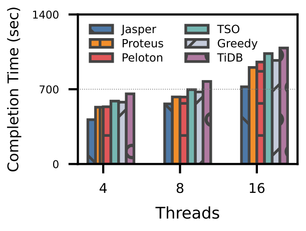
(b) CH-benchmark Balanced



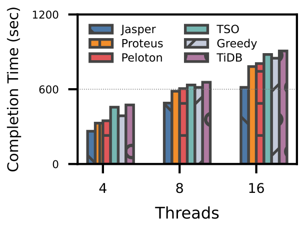
(c) CH-benchmark AP Heavy



(d) Twitter OLTP Heavy



(e) Twitter Balanced



(f) Twitter AP Heavy

Figure 6: Comparison of Workload Completion Time.

6.1.3 Metrics. We measure overall performance using **completion time**, which is the execution time to complete workloads, with the number of transactions executed by each client is fixed, following previous works [2]. In addition, we also measure the throughput of transactional queries executed in the row store (**OLTP throughput**) and average latency for analytical queries executed in the column store (**OLAP latency**). To better illustrate the trade-off between isolation and freshness. We use **H-Score**, which is a unified metric proposed by HyBench [61], quantifying the overall performance of HTAP systems by integrating QPS, TPS, hybrid throughput, and data freshness.

6.2 Performance Comparison

We measure the completion (execution) time to complete the OLTP-Heavy, Balanced, and OLAP-Heavy workloads. To conduct workload execution time experiments, we fix the number of OLAP transactions executed by each client following previous work [2]. We connect to TiDB with the number of clients set to 4, 8 and 16, and each client sends either TP or AP requests to TiDB according to the workload configuration.

Completion Time on Benchmarks. Figure 6 presents the workload completion times of various systems under three hybrid workload scenarios. Jasper consistently achieves the shortest completion times, with the most significant gains observed in the OLTP-Heavy setting. This performance advantage is attributed to

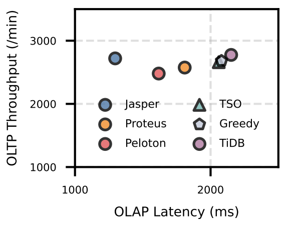


Figure 7: Throughput vs Latency on CH-Benchmark.

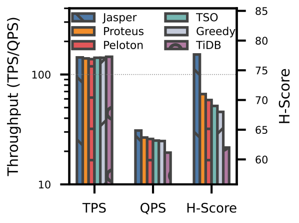
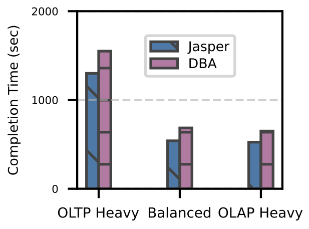
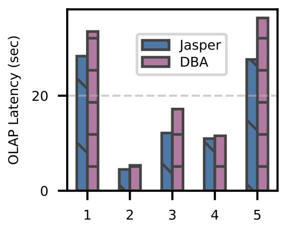


Figure 8: HyBench Performance and H-Score.

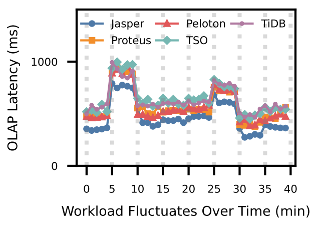


(a) Workload Completion Time

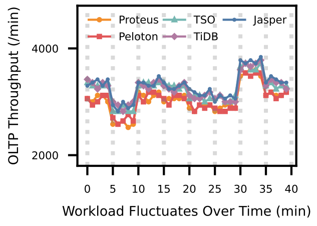


(b) OSS Query Latency

Figure 9: Performance Comparison of OSS Workload.



(a) OLAP Query Latency



(b) OLTP Throughput

Figure 10: Performance Comparison in Dynamic Workload. Every five minutes, we vary the ratio of transactions, query types, and the accessed partitions and ranges. The incremental update mechanism then determines whether the storage configuration should be adjusted.

Table 2: Overhead Analysis.

Method	First Call Time(s)	Subsequent Call Time(s)
Jasper (Ours)	348	6.7
Jasper-Optimizer	1432	40

its joint adaptive storage mechanism, which reduces data synchronization overhead and selects efficient partitioning strategies to accelerate query execution. In contrast, Proteus and Peloton exhibit longer completion times due to suboptimal partitioning and storage configurations, while TiDB and TSO perform worse overall, lacking an integrated optimization strategy for partitioning and columnar storage. Improper partitioning significantly increases query latency. These baselines partition strategies are heuristic and fail to utilize the partition pruning. Compared to other methods, Jasper reduces the average completion time by 20.72%–40.59% on the CH-benchmark and 20.43%–33.28% on the Twitter workload. These results highlight Jasper 's ability to improve the trade-off between workload isolation and data freshness in HTAP systems, effectively balancing both. Completion time increases as the proportion of TP transactions rises due to heavier update workloads and the resulting data synchronization overhead. Moreover, higher thread counts intensify resource contention, further degrading performance.

Throughput and Latency. To demonstrate the effectiveness of our approach in hybrid workloads, we evaluate OLAP latency under a balanced workload in the CH-benchmark and assess OLTP throughput using 20 clients. Figure 7 compares metrics across five methods. Jasper achieves the lowest latency, outperforming TiDB by 39.7%, and reducing latency by 28.39%, 19.9%, and 37.11% compared to Proteus, Peloton, and TSO, respectively. OLTP throughput remains comparable across all methods, as the primary bottlenecks in transactional performance are not directly tied to partitioning or storage format. A slight reduction in throughput for Jasper, Proteus, and Peloton is observed, attributed to vertical partitioning, which decomposes tables into multiple sub-tables, introducing overhead during insert and update operations. However, this partitioning strategy enables more efficient columnar storage. Overall, performance comparisons and ablation studies confirm that vertical partitioning offers overall benefits in hybrid workload scenarios.

We further investigate the performance with the HyBench benchmark [61]. Figure 8 shows the performance. Jasper consistently achieves the highest QPS due to its joint adaptive storage mechanism. Other methods exhibit lower QPS due to suboptimal partitioning and storage configurations. Jasper achieves the highest H-Score, indicating that our approach attains an optimal balance between data freshness and workload isolation.

Evaluation on TiDB Production Workload. We implement our approach on TiDB using real workloads OSS and compare its performance against the best configuration provided by DBAs. As shown in the Figure9a, we evaluate workload completion times under three scenarios: OLTP-Heavy, Balanced, and OLAP-Heavy. The results demonstrate that our method reduces execution time by 16.67% to 21.16% compared to the DBA-optimized configurations. Additionally, we present the latency of representative queries in Figure 9b, illustrating that our approach significantly reduces the latency of long-running queries.

6.3 Dynamic Workload Performance

To evaluate the effectiveness of Jasper's incremental update strategy, we construct a dynamic workload that changes every five minutes by adjusting the ratio of transactions, type of queries, accessed partitions and ranges. We allow Proteus, Peloton, and TSO to generate a new configuration online. As shown in Figure 10, configuration transformations of Jasper only occur in the periods with a significant pattern shift, such as at the 6, 11, 26, and 31-minute marks. The performance of all five methods fluctuates every five minutes in response to workload shifts. Overall, Jasper consistently achieves the lowest latency and highest throughput, maintaining its advantage across all workload phases. This is because Jasper rapidly adapts to workload shifts by incrementally generating optimized configurations, ensuring high performance.

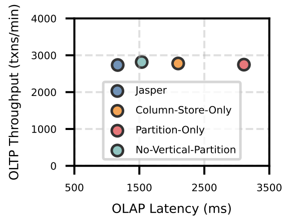


Figure 11: Comparison of Search Space.

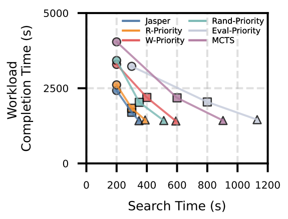
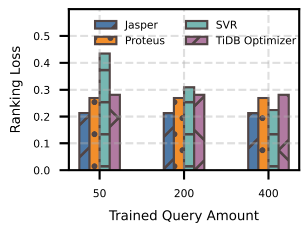
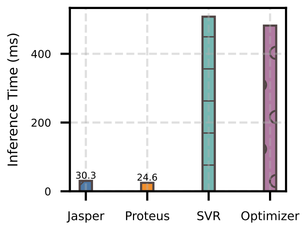


Figure 12: Comparison of Priority Assignments.



(a) Comparison of Accuracy



(b) Comparison of Inference Time

Figure 13: Analysis on Evaluation Model.

6.4 Analysis of Jasper

6.4.1 Overhead Analysis. We analyze the execution overhead of Jasper by measuring the time required for both initial and subsequent invocations under changing workloads, including search and evaluation phases. We also implement Jasper-Optimizer, a variant that invokes the query optimizer during each evaluation to generate physical plans. As shown in Table 2, the initial invocation incurs higher latency due to the complexity of searching from scratch across a large space. In contrast, subsequent calls under dynamic workloads are significantly faster, benefiting from incremental workload analysis that narrows the search space and improves efficiency. Jasper also outperforms Jasper-Optimizer in execution time, as the latter is dominated by overhead from repeated optimizer calls and connection setup. For training, Jasper uses a lightweight linear regression model, trained offline using data extracted from logs within a defined time window, incurring no additional data collection or runtime overhead.

6.4.2 Evaluation on Search Space Design. In these experiments, we conduct ablation studies to demonstrate the effectiveness of the joint search and MCST-HTAP. As shown in Figure 11, we compare Jasper with three alternative approaches: optimizing column-store-only, optimizing partitioning-only, and optimizing without vertical partition. We evaluate their performance in terms of OLAP latency and OLTP throughput. The experiments show that Jasper achieves lower latency while maintaining comparable throughput to the other methods. This demonstrates the effectiveness of jointly optimizing partitioning and column store. The Column-Store-Only and No-Vertical-Partition approaches show slightly higher latency, suggesting that both horizontal and vertical partitioning are necessary. The Partition-Only method produces the highest latency, emphasizing the importance of column store optimization.

6.4.3 Evaluation on Priority Metrics. We compare with different priority assignments on the search time and workload completion time. As show in Figure 12, R-Priority assigns priorities based on read frequency, and W-Priority does so based on write frequency. Higher read frequency results in higher priority, whereas higher write frequency leads to lower priority. Rand-Priority assigns priorities randomly, and Eval-Priority determines priorities according to the benefit estimated by the evaluation model. The results show that Jasper achieves the shortest search time to reach a similar completion time, whereas the other methods require substantially more time. Although Eval-Priority uses model-based priorities that are more accurate, its heavy reliance on the evaluation model incurs significant overhead, resulting in much longer search time. Overall, our priority strategies are able to identify high-quality configurations within a relatively short time, and all priority methods eventually converge.

6.4.4 Evaluation on Evaluation Model. We evaluate our evaluation model for estimating performance under different data layouts and compare its accuracy against Proteus's cost model, a learning-based SVR model [5], and the TiDB optimizer. Because these models output estimated query costs that may deviate from actual runtimes, we use ranking loss to assess accuracy, focusing on relative orderings rather than absolute values. We execute a variety of queries to obtain ground-truth runtimes, apply each model to predict costs, and compute ranking loss using Equation 11.

$$\text{Ranking Loss} = \frac{1}{|P|} \sum_{(i,j)\in P} \mathbb{I}\left[(\hat{y}_i > \hat{y}_j) \wedge (y_i < y_j)\right] \tag{11}$$

As shown in Figure 13a, we evaluate ranking loss under different amounts of training data. Jasper consistently yields the lowest loss. In contrast, Proteus and the TiDB optimizer show higher loss because their formula-based operator cost models lack parameter calibration and cannot capture synchronization overhead. The learning-based approach improves gradually as training data increases, reflecting its dependence on large datasets; with limited data, its accuracy remains low. Figure 13b further compares inference times: optimizer-independent methods like Jasper and Proteus achieve low latency, whereas learning-based and optimizer-dependent methods incur higher overhead due to repeated optimizer interactions, making them costly in multi-round searches.

7 CONCLUSION

In this paper, we present Jasper, a joint adaptive storage framework for HTAP systems. To effectively balance workload isolation and data freshness, Jasper performs integrated optimization of horizontal and vertical partitioning, along with selective column store configuration. To efficiently navigate the vast configuration space, we introduce MCTS-HTAP, a workload-aware search algorithm guided by a lightweight evaluation model that accounts for both query execution latency and data synchronization overhead. Furthermore, Jasper supports incremental storage reconfiguration, enabling the system to adapt swiftly to workload shifts without incurring significant performance disruption. Experimental results show that Jasper reduces HTAP workload completion time by 20.43% to 40.59%, demonstrating its effectiveness in addressing the isolation-freshness trade-off.

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.

[2] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Proteus: Autonomous adaptive storage for mixed workloads. In *Proceedings of the 2022 International Conference on Management of Data*. 700–714.

[3] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Tiresias: enabling predictive autonomous storage and indexing. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3126–3136.

[4] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 359–370.

[5] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[6] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1103–1114.

[7] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2017. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2017), 689–702.

[8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.

[9] Manoussos Athanassoulis, Kenneth Bøgh, and Stratos Idreos. 2019. Optimal column layout for hybrid workloads. *Proceedings of the VLDB Endowment* (2019).

[10] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 209–220.

[11] Nicolas Bruno and Rimma V Nehme. 2008. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 941–952.

[12] Zhuo Chang, Xinyi Zhang, Yang Li, Xupeng Miao, Yanzhao Qin, and Bin Cui. 2024. MFIX: An Efficient and Reliable Index Advisor via Multi-Fidelity Bayesian Optimization. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4343–4356.

[13] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *ACM SIGMOD Record* 27, 2 (1998), 367–378.

[14] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[15] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[16] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[17] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 418–431.

[18] Zhenghao Ding, Xinyi Zhang, Wei Lu, Wenlong Ma, Wenliang Zhang, and Xiaoyong Du. 2025. Promi: Progressive Live Migration in Distributed Database Systems. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 349–362.

[19] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S Sekeran, Fabián Rodriguez, and Laxmi Balami. 2018. Gridformation: Towards self-driven online data partitioning using reinforcement learning. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–7.

[20] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[21] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment* 4, 2 (2010), 105–116.

[22] MySQL Heatwave. 2021. Real-time analytics for MySQL database service. *Public documentation version* 3 (2021), 1–20.

[23] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2019. Towards learning a partitioning advisor with deep reinforcement learning. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[24] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 143–157.

[25] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[26] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[27] Jasper. 2025. Technical Report. https://github.com/Godroser/Jasper/blob/main/TechnicalReport.md.

[28] Alekh Jindal. 2010. The mimicking octopus: Towards a one-size-fits-all database architecture. In *VLDB PhD Workshop*. 78–83.

[29] Rong Kang, Shuai Wang, Tieying Zhang, Xianghong Xu, Linhui Xu, Zhimin Liang, Lei Zhang, Rui Shi, and Jianjun Chen. 2025. VIDEX: A Disaggregated and Extensible Virtual Index for the Cloud and AI Era. *arXiv preprint arXiv:2503.23776* (2025).

[30] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[31] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[32] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[33] Guoliang Li and Chao Zhang. 2022. HTAP databases: What is new and what is next. In *Proceedings of the 2022 International Conference on Management of Data*. 2483–2488.

[34] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. 2022. A resource-aware deep cost model for big data query processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 885–897.

[35] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1659–1674.

[36] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[37] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 37–50.

[38] Rimma Nehme and Nicolas Bruno. 2011. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1137–1148.

[39] OceanBase. 2025. OceanBase Features. https://en.oceanbase.com/docs/common-oceanbase-database-10000000001970964. Accessed: 2025-10-15.

[40] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal* 29, 1 (2020), 569–591.

[41] Zhicheng Pan, Yuanjia Zhang, Chengcheng Yang, Ahmad Ghazal, Rong Zhang, Huiqi Hu, Xiaoju Wu, Yu Dong, and Xuan Zhou. 2025. Hyper: Hybrid Physical Design Advisor with Multi-Agent Reinforcement Learning. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 1565–1578.

[42] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. 2007. Efficient use of the query optimizer for automated physical design. In *Proceedings of the 33rd international conference on Very large data bases*. 1093–1104.

[43] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. 2020. Fast and effective distribution-key recommendation for amazon redshift. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2411–2423.

[44] PingCAP. 2025. OSS Insight. https://github.com/pingcap/ossinsight. Accessed: 2025-7-15.

[45] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2014. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 97–112.

[46] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[47] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 558–569.

[48] Kexin Rong, Paul Liu, Sarah Ashok Sonje, and Moses Charikar. 2024. Dynamic Data Layout Optimization with Worst-case Guarantees. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4288–4301.

[49] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3290–3303.

[50] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[51] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 731–742.

[52] SingleStore. 2025. The Single Database for All Data-Intensive Applications. https://www.singlestore.com/solutions/real-time-analytics/.

[53] Haoze Song, Wenchao Zhou, Feifei Li, Xiang Peng, and Heming Cui. 2023. Rethink query optimization in htap databases. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[54] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.

[55] TiDB. 2025. Online DDL. https://docs.pingcap.com/tidb/stable/feature-online-ddl/.

[56] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. Polardb-imci: A cloud-native htap database system at alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[57] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A Bernstein. 2022. Budget-aware index tuning with reinforcement learning. In *Proceedings of the 2022 International Conference on Management of Data*. 1528–1541.

[58] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[59] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[60] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 193–208.

[61] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A new benchmark for HTAP databases. *Proceedings of the VLDB Endowment* 17, 5 (2024), 939–951.

[62] Xuanhe Zhou, Guoliang Li, Jianhua Feng, Luyang Liu, and Wei Guo. 2023. Grep: A graph learning based database partitioning system. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

[63] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyuan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. 2022. Autoindex: An incremental index management system for dynamic workloads. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2196–2208.

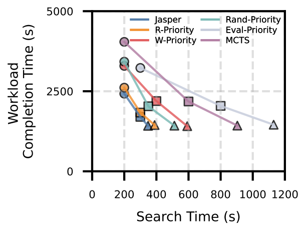
A EXTENDED EXPERIMENTS

A.1 Priority Analysis

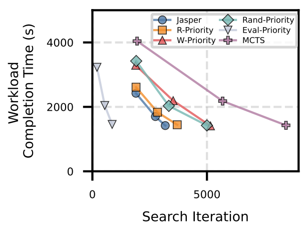
To quantify the correlation between priority and actual impact, we use the execution-time benefit estimated by our evaluation model as the baseline measure of impact for adding an action. We compare different priority assignment strategies: (1) R-Priority: higher read frequency yields higher priority; (2) W-Priority: higher write frequency yields lower priority; (3) Jasper: priorities derived from a combination of read and write frequency (our default); (4) Rand-Priority: priorities assigned uniformly at random; (5) Eval-Priority: priorities computed using the evaluation model's predicted execution-time benefit. We evaluate their performance in terms of the number of explored configurations and total search time.

As shown in Figure 14, to achieve the same level of performance, Eval-Priority explores the fewest configurations, followed by Jasper. However, Jasper achieves the smallest search time. This is because Eval-Priority repeatedly invokes the evaluation model to estimate the expected benefit of candidate actions, which introduces substantial computational overhead compared to Jasper's precomputed column-access statistics, resulting in significantly longer end-to-end search time. To further quantify the alignment between prioritization and action impact, we compute the Pearson correlation coefficient between the priorities assigned by each strategy and the action benefits estimated by Eval-Priority. As shown in Figure 15, Jasper exhibits the strongest correlation with Eval-Priority, confirming that the combined read/write–based priority assignment best approximates the relative importance of actions at negligible computational cost.

Our results show that when using weaker or mismatched priorities (e.g., W-Priority or Rand-Priority), the number of node expansions increases by 5.80-5.98x compared to Eval-Priority, but all strategies eventually converge to configurations with comparable workload completion time. This because although node pruning is applied based on the utility (computed from the priorities), MCTS-HTAP algorithm retains the ability to traverse the entire search space since the utility threshold θ is gradually reduced over time, as described in Section 4.3 of the original manuscript. Therefore, any mismatch in the initial priority assignment does not prevent convergence to the optimal configuration; it only affects the convergence speed.



(a) **Workload Completion Time vs. Search Time**



(b) **Workload Completion Time vs. Search Iteration**

Figure 14: Comparison of Different Priority Assignment Strategies.

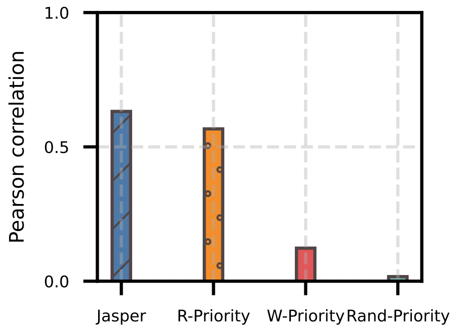


Figure 15: Correlation between each strategy's assigned priorities and the estimated action benefits. Higher correlation indicates stronger alignment with the estimated impact.

A.2 Sensitivity Analysis

We compare how different parameter settings affect the behavior of the workload-window mechanism. As shown in Table 3, we report the number of triggered searches, the number of reconfigurations, and the average throughput and latency normalized by the performance under the default setting. The results show a smaller *ws* significantly increases the number of searches but only slightly increases reconfigurations, as layout changes are applied only when the estimated benefit exceeds the cost. In contrast, a larger *ws* reduces the search frequency, ultimately resulting in worse performance. A similar trend is observed for δ: higher thresholds make Jasper less sensitive to workload changes, leading to fewer searches and degraded responsiveness.

Table 3: Sensitivity Analysis for Incremental Adjustment.

Method (*ws*=1min, δ=24% By Default)	Search Count	Transformation Count	Average TPS Norm.	Average Latency Norm.
Default	8	4	1	1
ws=20s	19	6	0.97	1.05
ws=10min	3	2	0.96	1.34
δ=10%	29	4	0.98	1.02
δ=50%	1	1	0.96	1.26

B DISCUSSION ON DIFFERENT ARCHITECTURES

B.1 Generality

Although Jasper is implemented on TiDB, its methodology is system-agnostic and applies to HTAP architecture that combines row storage with column-store replicas, rather than relying on TiDB's separated row–column design.

To demonstrate its generality and verify that the gains are not tied to TiDB-specific designs, we also implement Jasper and all baselines on OceanBase v4.3.5, which is an open-source HTAP system whose storage engine differs substantially from TiDB in two aspects: (1) OceanBase maintains baseline SSTables on disk and an incremental row-format MemTable, where all writes first land. Query execution merges the MemTable with SSTables to obtain the latest visible versions. (2) OceanBase can store SSTables in both row and column formats, but the two formats belonging to the same partition replica are co-located on a single OBServer node to preserve locality. This contrasts with TiDB's separated-engine architecture, where TiKV (row store) and TiFlash (column store) reside on different nodes.

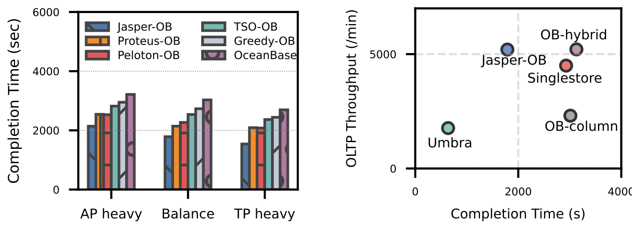
In OceanBase, when queries on column storage require the latest versions, the system reads incremental row data from the MemTable and merges them with the column-format SSTables, incurring additional merge overhead. We apply the MCTS-HTAP search methodology within OceanBase to explore effective partitioning strategies for SSTable and selectively configuring their column-store replicas. Using the same CH-Benchmark settings as in Section 6.2, we measure the workload completion time for OLTP-Heavy, Balanced, and OLAP-Heavy workloads. Figure **??** shows that Jasper-OB achieves the shortest completion time among all the baselines, with improvements ranging from 15.8% to 39.8%. This result is consistent with our observations on TiDB and suggests that the joint and adaptive optimization approach remains effective. These observations indicate that Jasper's benefits are not solely tied to TiDB-specific mechanisms and generalize to other HTAP architectures with different row–column freshness maintenance mechanisms.

Besides OceanBase, systems like Oracle [31] and SQL Server [32] adopt a unified architecture with disk-based row store and in-memory column store replicas. In these settings, Jasper's core idea naturally applies: jointly determine partitioning strategies and selectively configure column-store replicas for read-intensive partitions. As mentioned in Oracle [31]: "The contents of the IM column store are built from the persistent contents within the row store, a mechanism referred to as populate. It is possible to selectively populate the IM column store with a chosen subset of the database." Oracle supports applying the IM column store to multiple levels of granularity—tablespaces, tables, entire partitions, or even specific sub-partitions, which is compatible with Jasper's fine-grained selective column-store approach. Because Oracle and SQL Server are closed-source, we are unable to integrate Jasper's evaluation model with their optimizers and therefore cannot conduct end-to-end experiments. We leave the exploration of additional HTAP systems beyond OceanBase as an important direction for future work.

B.2 Comparison with Other Systems

We conduct comparison experiments with several HTAP databases in different architectures, including OceanBase-Hybrid [39], OceanBase-Column [39], SingleStore [52], and Umbra [30, 49]. OceanBase organizes data into baseline SSTables and an in-memory MemTable. In OceanBase-Column, SSTables are stored only in columnar format, whereas OceanBase-Hybrid maintains both row- and column-format SSTables for the same table. SingleStore, the successor to MemSQL, stores all data in column format while maintaining in-memory row indexes for transactional access. Umbra stores data primarily in a columnar format, and in its latest version [49], it keeps a portion of hot data in a row store. We also examined Hyrise [21], an in-memory column store database, but omit it from the evaluation since it fails to run serval complex-join queries in the CH-Benchmark. Since Umbra is a single-node system, we deploy all systems in single-node mode for a fair comparison. As TiDB's storage layer relies on Raft and cannot run standalone, we implement Jasper on an open-source version of OceanBase v4.3.5 (Jasper-OB) for this experiment.

We evaluate all systems under the CH-Benchmark in the Balanced workload scenario under the same experimental conditions as described in Section 6.2. As shown in Figure **??**, Jasper-OB achieves the best overall performance, with a completion time significantly lower than that of SingleStore and OB-Hybrid, owing to its adaptive storage mechanism. Both Jasper-OB and OB-Hybrid exhibit the highest throughput, which can be attributed to OceanBase's efficient LSM-tree–based data organization. Umbra attains the shortest workload completion time but suffers from low OLTP throughput, which is due to the columnar storage. Similarly, OB-Column also demonstrates limited throughput, as these columnar storage systems are generally less efficient under highly concurrent transactional workloads.



(a) Workload Completion Time in OceanBase

(b) Workload Completion Time and Throughput

Figure 16: Performance Comparison among Different Architectures.

C COST FUNCTIONS

Table 4 presents a comprehensive summary of the cost functions formulated for key operators. The cost functions rely on a set of tunable parameters to capture the performance characteristics of the underlying storage engine and target deployed system. To accurately estimate the execution overhead, our model incorporates both query-specific statistics—such as cardinality (rows) and tuple width (rowSize)—and system-specific performance coefficients (e.g., scanFactor, cpuFactor, and memFactor). By separating system-specific coefficients from query-specific statistics, the cost functions provide a flexible mechanism to quantify resource consumption across distinct operator types, ranging from I/O-intensive scans to CPU-intensive joins.

Table 4: Cost Function of Operators. The execution overhead of operators on different store is calculated using different Factor parameters.

Operator	Cost Function
TableScan	rows * log2(rowSize) * scanFactor + c * log2(rowSize) * scanFactor)
IndexScan	rows * log2(rowSize) * scanFactor
Selection	row * cpufactor * numFuncs
Update & Insert	rows * rowSize * netFactor
Sort	rows * log2(rows) * len(sort-items) * cpu-factor + memQuota * memFactor + rows * rowSize * diskFactor
HashAgg	rows * len(aggFuncs) * cpuFactor + rows * numFuncs * cpuFactor + (buildRows * nKeys * cpuFactor + buildRows * buildRowSize * memFactor + buildRows * cpuFactor + probeRows * nKeys * cpuFactor + probeRows * cpuFactor) / concurrency
SortMergeJoin	leftRows * leftRowSize * cpuFactor + rightRows * rightRowSize * cpuFactor + leftrows*numFuncs*cpuFactor + rightrows*numFuncs*cpuFactor
HashJoin	buildRows * buildFilters * cpuFactor + buildRows * nKeys * cpuFactor + buildRows * buildRowSize * memFactor + buildRows * cpuFactor + (probeRows * probeFilters * cpuFactor + probeRows * nKeys * cpuFactor + probeRows * probeRowSize * memFactor + probeRows * cpuFactor) / concurrency

D ONLINE RECONFIGURATION

The interaction between transformations and concurrently running queries/transactions is managed through an online reconfiguration process. (1) *Non-blocking copy phase.* When Jasper determines an new layout (L_{new}), the system initiates the creation of this layout in the background. Data from the existing layout (L_{old}) is asynchronously copied to L_{new} without blocking OLTP or OLAP traffic on L_{old}. (2) *Concurrency Maintenance.* While the copy is in progress, new updates (writes) continue to be applied to L_{old}. These updates are simultaneously tracked and applied to L_{new} via the Raft logs, ensuring L_{new} stays current. (3) *Atomic Switch.* Once L_{new} is fully caught up with L_{old} (i.e., they are consistent up to a certain commit timestamp), the system performs a rapid, atomic metadata update. This switch redirects the logical view of the data segment from L_{old} to L_{new}. This guarantees that ongoing queries always observe a consistent snapshot throughout the entire transformation process. This process is similar to online schema evolution in TiDB [55].